

SOCKET PROGRAMMING WITH PYTHON

**TCP/UDP/IP
HTTP, SMTP, FTP, TELNET
CGI MODULE
APACHE ON LINUX/WINDOWS
FORK/THREADED SERVER
PRISTINE ENVIRONMENT**



AJIT SINGH

COPYRIGHT (C) 2019, AJIT SINGH.

Preface

This book will introduce you to the Python Socket programming. It's aimed at building socket program , but even if you've written programs in Python before and want to add Python Socket programming to your list of skill sets, this will surely help you a lot.

This book is about using Python to get the socket program done on Windows as well as LINUX.

I hope by now you have heard of Python, the exciting object-oriented scripting language that is rapidly entering the programming mainstream. Although Python is perhaps better known on the Unix platform, it offers a superb degree of integration with the Windows environment. One of us, Mark Hammond, is responsible for many of Python's Windows extensions and has co-authored the Python COM support, both of which are major topics of this book. This book can thus be considered the definitive reference to date for Python on the Windows platform.

This is intended to be a practical book focused on several examples of socket programs. It doesn't aim to teach Python programming, although i do provide a brief tutorial. Instead, it aims to cover socket programming.

I also include some lab assignments to get more familiarize with socket programming.

NOTE

Sometimes, I'll include a note such as this when something might be confusing or there's a more appropriate Pythonic way to do it.

Audience

This book is for anybody interested in learning what seems to be emerging as the world's most popular computing language, whether or not you have learned any programming before as well as the aspirants of network professional.

Contents

1	Introduction	4
2	Basic Socket Overview	5
2.1	Creating a socket	
2.2	Connecting a socket and data transfer	
2.3	Binding a name to socket	
2.4	Listening and accepting connections	
2.5	UDP sockets	
2.6	Closing the socket	
2.7	Using functions provided in socket module	
	2.7.1 Functions based on resolver library	
	2.7.2 Service-related functions	
	2.7.3 Miscellaneous functions	
3	Basic network structures design	9
3.1	Designing a TCP server	
3.2	The TCP client	
3.3	Modeling datagram applications	
		13
4	Advanced topics on servers	
4.1	Building a pristine environment	
4.2	Handling multiple connections	
	4.2.1 Threaded servers	
	4.2.2 Using select	
	4.2.3 Fork servers	
4.3	Dealing with classes	
	4.3.1 Simple connection object	
	4.3.2 Applying a design pattern	
4.4	Advanced aspects concerning clients	
5	HTTP protocol	21
5.1	CGI module	
	5.1.1 Build a simple CGI script	
	5.1.2 Using CGI module	
	5.1.3 Configuring Apache on Linux for using with CGI scripts	
6	Common protocols	26
6.1	Designing Telnet applications	
6.2	File Transfer Protocol	
6.3	SMTP protocol	
7	Getting Started With Python In Windows	30

Chapter 1: Introduction

Network programming is a buzzword now in the soft world. We see the market lled with an avalanche of network oriented applications like database servers, games, Java servlets and applets, CGI scripts, di erent clients for any imaginable protocol and the examples may con-tinue. Today, more then half of the applications that hit the market are network oriented. Data communication between two machines (on local net or Internet) is not any more a curiosity but is a day to day reality. "The network is the computer" says the Sun Microsystem's motto and they are right. The computer is no more seen as a separate entity, dialoging only with it's human operator but as part of a larger system - the network, bound via data links with other thousands of other machines.

This book is presenting a possible way of designing network-oriented applications using Python. With a little effort, the examples are portable to most of the operating system. Presenting a quick structure of this book, first four sections are dealing with primitive design { at socket level } of network applications. The remaining sections are treating specific protocols like http, ftp, telnet or smtp. The section dealing with http will contain a subsection about writing CGI scripts and using the cgi module.

Going further on more concrete subjects, i am going to analyze the possibilities of network programming provided in Python. Raw network support is implemented in Python through the socket module, this module comprising mostly of the system-calls, functions and constants defined by the 4.3BSD Interprocess Communication facilities (see [1]), implemented in object-oriented style. Python offers a simple interface (much simpler than the corresponding C implementation, though based on this one) to properly create and use a socket. Primarily, is defined the socket() function returning a socket object. The socket has several methods, corresponding to their pairs from C sys/socket.h, like bind(), connect(), listen() or accept(). Programmers accustomed with socket usage under C language will find very easy to translate their knowledge in the more-easy-to-use socket implementation under Python. Python eliminates the daunting task of filling structures like sockaddr in orhostent and ease the use of previously mentioned methods or functions { parameter passing and functions call are easier to handle. Some network-oriented functions are provided too: gethostbyname(), getprotobyname() or conversion functions ntohs(), htons(), useful when converting integers to and from network format. The module provides constants like SOMAXCONN, INADDR *, used in gesockopt() or setsockopt() functions. For a complete list of above mentioned constants check your UNIX documentation on socket implementation.

Python provide beside socket, additional modules (in fact there is a whole bundle of them) supporting the most common network protocols at user level. For example we may find useful modules like httpplib, ftplib, telnetlib, smtpplib. There is implemented support for CGI scripting through cgi module, a module for URL parsing, classes describing web servers and the examples may continue. This modules are specific implementations of well known protocols, the user being encouraged to use them and not trying to reinvent the wheel. I hope that you will enjoy the richness of Python's network programming facilities and use them in new and more exciting ways on LINUX as well WINDOWS Environment.

Because all the examples below are written in Python, the reader is expected to be fluent with this programming language.

Chapter 2: Basic Socket Overview

The socket is the basic structure for communication between processes. A socket is defined as "an endpoint of communication to which a name may be bound" [1]. The 4.3BSD implementation defines three communication domains for a socket: the UNIX domain for on-system communication between processes; the Internet domain for processes communicating over TCP(UDP)/IP protocol; the NS domain used by processes communicating over the old Xerox communication protocol.

Python is using only the first two communication domains: UNIX and Internet domains, the AF_UNIX and AF_INET address families respectively. UNIX domain addresses are represented as strings, naming a local path: for example /tmp/sock. This can be a socket created by a local process or, possibly, created by a foreign process. The Internet domain addresses are represented as a(host, port) tuple, where host is a string representing a valid Internet hostname, say matrix.ee.utt.ro or an IP address in dotted decimal notation and port is a valid port between 1 and 65535. It is useful to make a remark here: instead of a qualified hostname or a valid IP address, two special forms are provided: an empty string is used instead of INADDR_ANY and the '<broadcast>' string instead of INADDR_BROADCAST.

Python offers all five types of sockets defined in 4.3BSD IPC implementation. Two seem to be generally used in the vastness majority of the new applications. A stream socket is a connection-oriented socket, and has the underlying communication support the TCP protocol, providing bidirectional, reliable, sequenced and unduplicated flow of data. A datagram socket is a connectionless communication socket, supported through the UDP protocol. It offers a bidirectional data flow, without being reliable, sequenced or unduplicated. A process receiving a sequence of datagrams may find duplicated messages or, possibly, in another order in which the packets were sent. The raw, sequenced and reliably delivered message socket types are rarely used. Raw socket type is needed when one application may require access to the most intimate resources provided by the socket implementation. Our document is focusing on stream and datagram sockets.

Creating a socket

A socket is created through the `socket(family, type [, proto])` call; family is one of the above mentioned address families: AF_UNIX and AF_INET, type is represented through the following constants: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET and SOCK_RDM. proto argument is optional and defaults to 0. We see that `socket()` function returns a socket in the specified domain with the specified type. Because the constants mentioned above are contained in the socket module, all of them must be used with the `socket.CONSTANT` notation. Without doing so, the interpreter will generate an error. To create a stream socket in the Internet domain we are using the following line:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Substituting `socket.SOCK_STREAM` with `socket.SOCK_DGRAM` we create a datagram socket in the Internet domain. The following call will create a stream socket in the UNIX domain:

```
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
```

I discussed so far about obtaining a socket of different types in different communication domains.

Connecting a socket and data transfer

A server from our point of view is a process which listen on a specified port. We may call the association port, process as a service. When another process wants to meet the server or use a specific service it must connect itself to the address and portnumber specified by the server. This is done calling the socket method `connect(address)`, where address is a pair (host, port) in the Internet domain and a pathname in the UNIX domain. When using the Internet domain a connection is realized with the following code:

```
sock.connect(('localhost', 8000))
```

while in UNIX domain,

```
sock.connect('/tmp/sock')
```

If the service is unavailable or the server don't want to talk with the client process a `socket.error-(111, 'Connection refused')` is issued. Elsewhere, after the connection is established with the desired server, data is sent and received with `send(buffer [, flags])` and `recv(buffer [, flags])` methods. These methods accepts as mandatory parameter the size of the buffer in bytes and some optional flags ; for a description about the meaning of the args consult the UNIX man page for the corresponding function.

Binding a name to socket

The socket, after creation, is nameless, though it have an associated descriptor. Before it can be used it must be bind to a proper address since this is the only way a foreign process may reference it. The `bind(address)` method is used to "name" a socket. The meaning of the address is explained above. Next call will bind a socket in the Internet domain with address composed from hostname localhost and port number 8000 :

```
sock.bind(('localhost', 8000))
```

Please take care when typing: indeed there are two pairs of parenthesis. Doing elsewhere the interpreter will issue a `TypeError`. The purpose of the two pairs of parenthesis is simple: address is a tuple containing a string and an integer. The hostname must be properly picked, the best method is to use `gethostname()` routine in order to assure host independence and portability. Creating a socket in the UNIX domain use address as a single string, naming a local path:

```
sock.bind('/tmp/sock')
```

This will create the `'/tmp/sock'` le (pipe) which will be used for communication between the server and client processes. The user must have read/write permissions in that specific directory where the socket is created and the le itself must be deleted once it's no longer of interest.

Listening and accepting connections

Once we have a socket with a proper name bound to it, next step is calling the `listen(queue)` method. It instructs the socket to passively listen on port port . `listen()` take as parameter an integer representing the maximum queued connection. This argument should be at least 1 and maximum, system-dependent, 5. Until now we have a socket with a proper bounded address. When a connection request arrives, the server decide whether it will be accepted or not. Accepting a connection is made through the `accept()` method. It takes no parameter but it returns a tuple (clientsocket, address) where clientsocket is a new socket server uses to communicate with the client and address is the client's address. `accept()` normally blocks until a connection is realized. This behavior can be overridden running the

method in a separate thread, collecting the new created socket descriptors in a list and process them in order. Meantime, the server can do something else. The above mentioned methods are used as follows:

```
sock.listen(5)
clisock, address = sock.accept()
```

The code instructs the socket on listening with a queue of five connections and accept all incoming "calls". As you can see, `accept()` returns a new socket that will be used in further data exchanging. Using the chain `bind-listen-accept` we create TCP servers. Remember, a TCP socket is connection-oriented; when a client wants to speak to a particular server it must connect itself, wait until the server accepts the connection, exchange data then close. This is modeling a phone call: the client dial the number, wait till the other side establish the connection, speak then quit.

UDP sockets

We chose to deal with connectionless sockets separately because these are less common in day to day client/server design. A datagram socket is characterized by a connectionless and symmetric message exchange. Server and client exchange data packets not data streams, packets flowing between client and server separately. The UDP connection resemble the postal system: each message is encapsulated in an envelope and received as a separate entity. A large message may be split into multiple parts, each one delivered separately (not in the same order, duplicated and so on). Is the receiver's duty to assemble the message.

The server have a `bind()` method used to append a proper name and port. There are no `listen()` and `accept()` method, because the server is not listening and is not accepts connection. Basically, we create a P.O.Box where is possible to receive messages from client processes. Clients only send packets, data and address being included on each packet.

Data packets are send and received with the `sendto(data, address)` and `recvfrom(buffer [, flags])` methods. First method takes as parameters a string and the server address as explained above in `connect()` and `bind()`. Because is speci ed the remote end of the socket there is no need to connect it. The second method is similar to `recv()`.

Closing the socket

After the socket is no longer used, it must be closed with `close()` method. When a user is no more interested on any pending data a shutdown may be performed before closing the socket. The method is `shutdown(how)`, where `how` is: 0 if no more incoming data will be accepted, 1 will disallow data sending and a value of 2 prevent both send and receive of data. Remember: always close a socket after using it.

Using functions provided in socket module

It is presented before that socket module contains some useful functions in network design. This functions are related with the resolver libraries, `/etc/services` or `/etc/protocols` mapping les or conversions of quantities.

Functions based on resolver library

There are three functions using `BIND8` or whatever resolver you may have. This functions usu-ally converts a hostname to IP address or IP address to hostname. One function is not related with the resolver, `gethostname()`, this function returning a string containing the name of the machine where the script is running. It simply read (through the corresponding C function) the `/etc/HOSTNAME`. We can use (prior to version 2.0) the `getfqdn(hostname)` function, which return the "Fully Qualified Domain Name" for the hostname . If the parameter is missing it will return the localhost's FQDN. Two functions are used to translate hostnames into valid IP addresses: `gethostbyname(hostname)` and `gethostbyname ex(hostname)`. Both functions take as mandatory parameter a valid

hostname. First function returns a dotted notation IP address and the last function (ex from "extended") a tuple (hostname, aliaslist, ipaddrlist). Last function discussed here is gethostbyaddr(ipaddr) returning host-name when the IP address is given.

Service-related functions

/etc/services is a file which maps services to portnumbers. For example, http service is mapped on port 80, ftp service on port 21 and ssh service on port 22. getservbyname(servname) is a function that translates a service name into a valid portnumber, based on the file presented above. The method will assure platform independence (or even computer independence) while the same service may not be mapped exactly on the same port.

When we want to translate a protocol into a number suitable for passing as third argument for the socket() function use getprotobyname(proto name). It translates, based on /etc/protocols file, the protocol name, say GGP or ICMP, to the corresponding number.

Miscellaneous functions

To convert short and long integers from host to network order and reversed, four functions were provided. On some systems (i.e. Intel or VAX), host byte ordering is different from network order. Therefore programs are required to perform translations between these two formats. The table below synthesizes their use:

Function Name	Synopsis
htons(sint)	Convert short integer from host to network format
ntohs(sint)	Convert short integer from network to host format
htonl(lint)	Convert long integer from host to network format
ntohl(lint)	Convert long integer from network to host format

Chapter 3: Basic network structures design

This chapter is focused in presenting four simple examples to illustrate the previous explained network functions and methods. Examples are providing a simple code designed for clarity not for efficiency. There are presented two pairs of server-client structures: one pair for the TCP protocol and the other one for the UDP protocol. The reader is encouraged to understand these examples before going further.

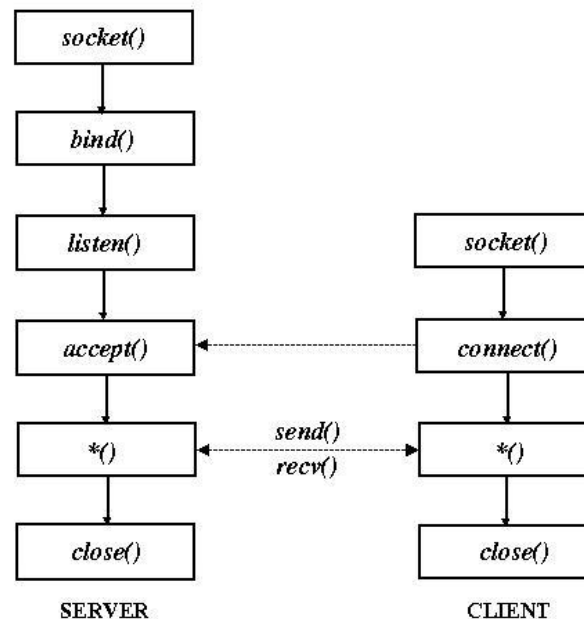


Figure 1: TCP connection

A stream connection is presented above, figure 1. Observe how the processes interact: the server is started and must be in accept state before the client issues its request. The client's `connect()` method is trying to rendezvous with the server while this one is accepting connections. After the connection has been negotiated, follows a data exchange and both sides call `close()` terminating the connection. Remember, this is a one connection diagram. In the real world, after creating a new connection (in a new process or in a new thread) the server returns to accept state. `*(*)` functions are user-defined functions to handle a specific protocol. Data transfer is realized through `send()` and `recv()`.

A datagram exchange is presented in figure 2. As you can see, server is the process which binds itself a name and a port, being the first that receives a request. The client is the process which first sends a request. You may insert a `bind()` method into client code, obtaining identical entities. Which is the server and which is the client it is hard to predict in this case.

Hoping that the presented diagrams were useful, we are going further presenting the code on four simple structures, modeling the TCP and UDP client-server pairs.

Designing a TCP server

Designing a TCP server must follow the bind-listen-accept steps. Producing code for a simple echo server, running on localhost and listening on port 8000 is fairly simple. It will accept

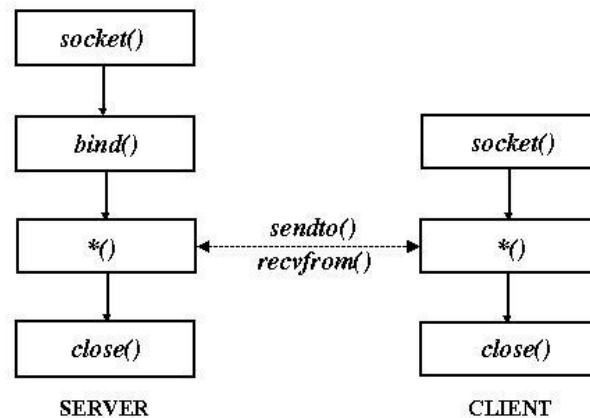


Figure 2: UDP connection

a single connection from the client, will echo back all the data and when it receives nothing close the connection.

```

import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8000))
serversocket.listen(1)
clientsocket, clientaddress = serversocket.accept() print
'Connection from ', clientaddress
while 1:
    data = clientsocket.recv(1024) if
    not data: break
    clientsocket.send(data)
clientsocket.close()
  
```

On first line is declared the import statement of the socket module. Lines 3 through 6 is the standard TCP server chain. On line 6 the server accept the connection initiated by the client; the accept() method returns clientsocket socket, further used to exchange data. The while 1 loop is a the "echo code"; it sends back the data transmitted by the client. After the loop is broken (when the client send an empty string and the if not data: break condition is met) the socket is closed and program exits.

Before getting into client code, a quick way to test the server is a telnet client. Because telnet run over TCP protocol there should be no problem in data exchange. Telnet will simply read data from the standard input and send it to server and, when it receives data, display it on the terminal.

```

$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost
Escape character is ^].

$
  
```

After the telnet send some data it enter in terminal mode: type something and the server will echo it back.

The TCP client

It is time to focus on writing client's code for our echo server. The client will enter a loop where data is read from the standard input and sent (in 1024 bytes packets - this is the maximum amount of data the server is reading specified in the server code above) to the server. The server is sending back the same data and the client is posting it again. Here is the code:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8000))
while 1:
    data = raw_input('>')
    clientsocket.send(data) if
    not data: break
    newdata = clientsocket.recv(1024)
    print newdata
clientsocket.close()
```

The client first send data and only after the data was sent, in case data is zero, exits. This prevent server from hanging (remember, the server exit when it receive an empty string). We decide to store the received data in newdata to prevent any problems that may occur, if no data or something wrong was transmitted back from the server. Compared with the server the client code is much simpler - only the socket creation and a simple connect, then the protocol follows.

Modeling datagram applications

Was specified before that is a slightly difference between a datagram client and a datagram server. Let's consider again the example of a server running on localhost and receiving packets on port 8000 .

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('localhost', 8000))
while 1:
    data, address = recvfrom(1024) if
    not data: break; sock.sendto(data,
    address)
sock.close()
```

As you can see in the example there are no listen or accept steps. Only binding is necessary, elsewhere the client is not aware where it must send packets to server. In simple terms, the kernel simply 'push' data packets to the server on the specified port, being no need for con rmation of connection and so on. The server decide if the packet will be accepted or not. recvfrom() returns beside data the client's address, used further in the sendto() method. The client is not much different from the server:

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while 1:
    data = raw_input('>') sock.sendto(data,
    ('localhost', 8000)) if not data: break;

    newdata = sock.recvfrom(1024)
    print newdata
sock.close()
```

We do not have a bind method, while the server is using address returned by `recvfrom()` method. Is possible to design an almost symmetric UDP server/client, using the same structure. The client will be the side that will initiate first the connection.

Chapter 4: Advanced topics on servers

This section cover some advanced techniques extensively used in modern server designs. The four structures previously presented are just some illustrations on how to use socket-related functions. The reader is not encouraged to use this four structures, because the code is very limited and was produced for exempli cation purpose only. Going further, is necessary to learn some well-known patterns, enhancing server design.

Building a pristine environment

When a server is started it must define some parameters needed at run time. This is done through `get*()` functions. Next code is written for a web server and is trying to determine the system's hostname, port number on which http protocol is running.

```
hostname = gethostname()
try:
    http_port = getservbyname('http','tcp') except
socket.error:
    print 'Server error: http/tcp: unknown service'
```

We check `getservbyname()` against errors because is possible for a service to be absent on that system. Definitely, on new implemented services/protocols is useless to call this function. `hostname` and `http_port` will be further used as parameters to `bind()`.

After the previous step is over, the server must be disassociated from the controlling terminal of it's invoked, therefore it will run as demon. In C this is done with the `ioctl()` system-call, closing the standard input, output and error and performing some additional operations [1]. In Python the server must be invoked with a "&" after its name. Another way to complete this, the recommended one, is to call a `fork()` which create two processes: the parent, that exits immediately and the child (the child is our application). The child is adopted by `init`, thus running as demon. This is an affordable method and is not implying complicate code:

```
import os

...

pid = os.fork()
if pid: os._exit(0) else:

    ...    #server code begin here
```

The server will be able no more to send messages through the standard output or standard error but through `syslog` facility. After this two steps are completed, we can select a method for handling multiple connections in the same time.

Handling multiple connections

Analysing the servers presented above, sections 3.1 and 3.3, it's clear that the design has a huge shortcoming: the server is not able to handle more than one client. A real server is designed to serve more then one client and moreover, multiple clients in the same time.

There are three methods to handle this issue: through the `select()` functions, creating a new process for each incoming request via `fork()` or to handle that request on a separate thread. Threading is the most elegant method and we recommend it. Using `select()` save some CPU in case of heavily accessed servers and may be a good option sometime. Creating a new process for each incoming connection is the most used pattern in current

server design (for example Apache use it) but it have the disadvantage of wasting CPU time and system resources. Again, we recommend using threaded servers.

Threaded servers

Threaded servers use a separate thread for handling each connection. Threads are defined as a light processes and are running in and along with the main process which started them. The diagram is presented below:

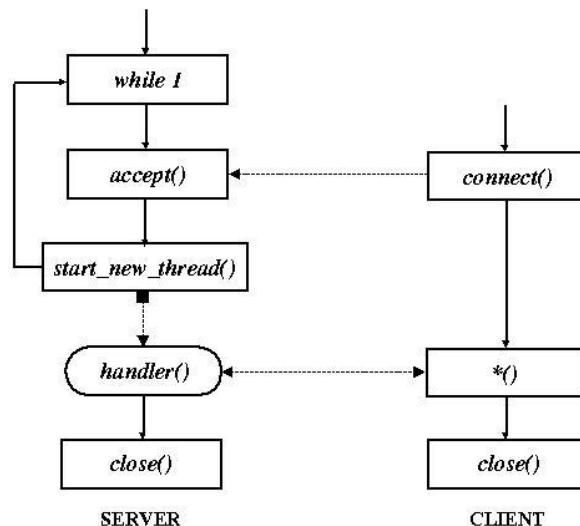


Figure 3: Threaded server diagram

Assuming we have a user-defined function called handler to handle a simple connection, each time a client wants to exchange data with the server handler is started on a separate thread doing its job. The squared and dashed arrow denote a new thread creation. The new thread will interact further with the client. Bellow is provided an example for a TCP server which main socket is called sock.

```

import socket, thread

def handler(socket):
    ...
    ...

while 1:
    clisock, addr = sock.accept()
    syslog.syslog('Incoming connection')
    thread.start_new_thread(handler, (clisock,))

```

The handling function must be defined before being passed as argument for the new thread. In example this function take as parameter the socket corresponding to the client which initiate the connection. The function may perform any kind of operation on that socket. The logging job is no more done through the print function but through syslog. Alternatively, is possible to write a function to handle this job separately (producing our own logs in user-defined les).

Another choice is to use the Threading module which offers a more exible implementation of threads. In fact, you create a thread object, with several methods, more easy to handle. The function's arguments must be passed in a tuple (or if it is a single element use a singleton [4]) in order to have an error-free code, elsewhere the interpreter will issue a TypeError. The design may be enhanced putting the accept() method in a separate thread; consequently the server will

be free (if `accept()` blocks, the normal behavior, it will do this separate from the main thread of the server) and may be used to perform some additional operation or even listen to another port.

I suggest you the design of a web server running on standard HTTP port and being remotely econtrolled in the same time on an arbitrary port. Other valuable feature: using threads add a professional "touch" to your code and eliminate the need of write code to handle fork system-call and additional functions. In the same time, if you want to design the web server mentioned above, there will be at least three processes running in the same time, causing processor to slow down.

Using select

Python provides a straightforward implementation of the UNIX standard `select()` function. This function, defined in `select` module, is used on multiplexing I/O requests among multiple sockets or le descriptors. There are created three lists of socket descriptors: one list with descriptors for which we want to read from, a list of descriptors to which we want to be able to write and a list which contains sockets with 'special conditions' pending. The only "special condition" implemented is the out-of-band data, specified in the socket implementation as `MSG_OOB` constant and used as special `ag` for the `send()` and `recv()` methods. This three lists are passed as parameters to function beside a timeout parameter, specifying how long should wait `select()` until returns, in case no descriptor is available. In return, `select()` provide us with three lists: one lists with socket which might be read from, another with writable sockets and the last one corresponding to 'special condition' category.

A special socket feature will help in a more exible approach to handle `select()` calls. This is the `setblocking()` method, which, if called with a 0 parameter, will set the nonblocking option for a socket [7]. That is, requests that cannot complete immediately, causing the process to suspend aren't executed, but an error code is returned. There is provided a code for a better understanding of the subject:

```
import socket, select

preaders = []
pwriters = []

...

sock.setblocking(0)
preaders.append(sock)

rtoread, rtowrite, pendingerr = select.select(preaders, pwriters, [], 30.0)

...
```

As you can see we constructed two lists, one for the potential readers and the other ones for the potential writers. Because there is no interest in special condition, an empty list is passed as parameter to `select()` instead of a list with sockets. The timeout value is 30 seconds. After the function returns, we have two usable lists, with the available sockets for reading and writing. Is a good idea at last to add the main server socket (here `sock`) to the potential readers list, therefore the server being unable to accept incoming requests. Accepting requests mechanism is fairly simple: when a request is pending on the server socket, it is returned in the ready to read list and one may execute an `accept()` on it. The resulting client socket may be appended to any of the lists (may be on both, depending on protocol). When timeout parameter is 0 the `select()` takes the form of a poll, returning immediately.

Fork servers

Fork servers are the first choice in network design today. One example will give you a birds-eye view: Apache, the most popular web server today is conceived as a fork server. Fork servers use the exible process management implemented on most modern UNIX flavors.

The basic idea is to get a new process for every incoming request, the parent process only listen, accept connections and "breed" children. Using the fork alternative you must accept its shortcomings: you must deal with processes at professional level, the server is much slower compared to a server designed with threads and is a bit more complicated. This implies that you must design a remain function to collect the zombies, adding separate code for parent and for children and so on. In figure 4 is presented a diagram showing the algorithm for a fork server. In the figure is not presented the remain function; usually it is called before calling `fork()`. The code for parent and for children is explicitly marked. The creation of a new process is signaled through a circled and dashed arrow. After the data exchange is over and both parts call `close()` on socket, the child is waited by the parent through a `wait()` or `waitpid()` system-call. This system-calls prevents the birth of so-called zombies. It is customary to create a list where to add all the children pids resulted after the calling `fork()`. This list is passed as parameter to the remain function which take care of them. Are presented two fork methods:

simple fork: this method simply call `fork()` then with an if-then-else, based on the resulting pid execute separate code for parent and for child. If the child is not waited it will become a zombie.

separate fork: the parent forks and create the child one. This forks again and create child two then exit. Child one is waited in the parent. Because child two was left without a parent it is adopted by init, thus, when it call `exit()`, init will take care of this process not to become a zombie.

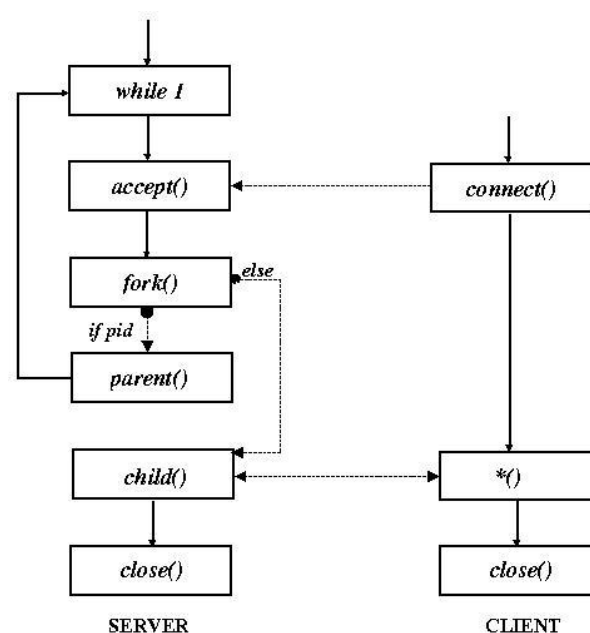


Figure 4: Fork server diagram

I present now code for a fork server with both methods. First method is usually expected. Again, sock is the server's socket.

```

import os, socket, syslog

children_list = []

def fireman(pids):
    while children_list:
        pid, status = os.waitpid(0, os.WNOHANG) if

```



```

        not pid: break
        pids.remove(pid)

def handler(socket):
    ...

...

while 1:
    clisock, addr = sock.accept()
    syslog.syslog('Incoming connection')
    fireman(children_list)
    pid = os.fork()

    if pid: #parent
        clisock.close()

        children_list.append(pid)

    if not pid: #child
        sock.close()
        handler(clisock)
        os._exit(0)

```

handler function is described in 4.2.1. Please see how separate code is executed for parent and for child. It is a good habit to close the server socket sock in the child and the client socket clisock in the parent. The fireman wait all the children from the specified list, passed as parameter. The second method may be implemented as follows:

```

...

while 1:
    clisock, addr = sock.accept()
    syslog.syslog('Incoming connection')
    fireman(children_list)
    pid = os.fork()

    if pid: #parent
        clisock.close()
        children_list.append(pid)

    if not pid: #child1
        pid = os.fork()
        if pid: #child2
            sock.close()
            handler(clisock)
            os._exit(0)
        else:
            os._exit(0)

```

First child only forks then exit. It is waited on parent, so there must be no problem on it. The second child is adopted and then it enter in "normal" mode, executing the handler function.

Dealing with classes

This section describes how to do real object-oriented programming. Because it is a common topic, both referring server and clients, we decided to cover both aspects in this section. You must learn how to construct classes that deals with networking functions, constructing connection objects and much more. The purpose is to design all this objects after a given

pattern [3]. This will simplify the design and will add more reusability and power to your code. For more information about classes or using classes in Python, see [5] and [6].

Simple connection object

This section's goal is to design a simple connection object. The purpose of designing such an object is to ease the design of a given application. While you have already designed the proper components (objects) you may focus on designing the program at a higher level, at component level. This give you a precious level of abstraction, very useful in almost all the cases.

Let's suppose we want to create a TCP socket server in one object. This will allow connection from TCP clients on the speci ed port. First, there must be de ned several methods to handle the possible situation that appear during runtime. Referring at figure 1, we see that the server must "launch" a socket and listen on it, then, when a connection is coming, accepting the connection then deal with it. The following methods are visible to the user: OpenServer, EstablishConnection, CloseConnection. OpenServer is a method that sets the hostname and the port on which the server is running. When a connection is incoming, EstablishConnection decide if it will be accepted based on a function passed as parameter. CloseConnection is self explanatory.

```
import socket

...

class TCPServer:

    def __init__(self): pass;

    def OpenServer(self, HOST="", PORT=7000): try:
        sock = socket.socket(socket.AF_INET,\
                             socket.SOCK_STREAM);
    except socket.error:
        print 'Server error: Unable to open socket'
        sock.bind((HOST, PORT));
        sock.listen(5);
        return sock;

    def EstablishConnection(self, sock, connection_handler): while 1:

        client_socket, address = sock.accept(); print
        'Connection from', address';
        thread.start_new_thread(connection_handler,\
                                (client_socket,));

    def CloseConnection(self, sock):
        sock.close();
```

The `__init__()` function does nothing. When the server is started, the `TCPServer` object, through `OpenServer`, takes as parameters the hostname on where it run and the service portnumber. Alternatively, this may be done passing this parameters to the `__init__()` function. The method perform also an error checking on socket. We are going to insist over the `EstablishConnection` method. It implement a threaded server and takes, among other parameters, a function name (here called `connection_handler`) used to handle the connection. A possible enhancement is to design this method to take as parameter a tuple (or a dictionary) containing the `connection_handler` parameters, which is supplied in our example with one parameter, generated inside the method which called it, the `client_socket`. There is no sense to insist over `CloseConnection` method. It was provided to assure a uniform interface to the `TCPServer` object. Future implementation (through inheritance or through object composition, see [3]) may find

useful to specify additional features to this method, but now the ball is in your field.

Applying a design pattern

The state design pattern is our purpose. The design will include five classes: one class, the TCPServer, will be the client (the user of the other four classes); TCPConnection is a class that realizes a TCP connection: through OpenConnection(), HandleConnection() and CloseConnection() it manages the socket; also, are created three state classes, representing the states in which TCPConnection may be. These classes are inherited from an abstract class, called TCPState. Below is presented a diagram illustrating the relationship between the connection object and its states (represented as classes): Here is the code for the TCPConnection

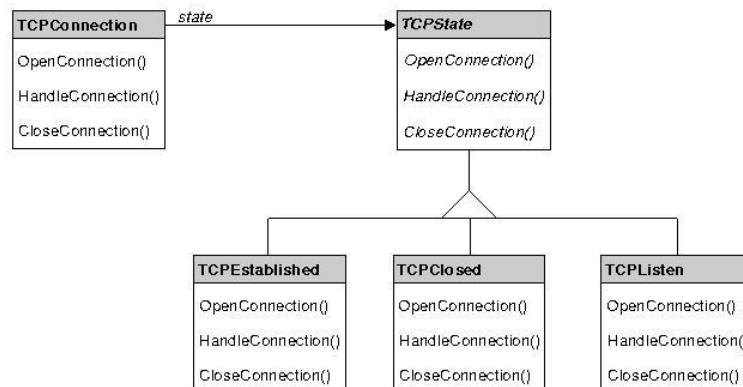


Figure 5: Designing a TCP connection with state pattern

class. The TCPServer class (please be careful: when the TCPServer example was given in section 4.3.1, it was a server; here we design only a connection class, which will further be used in designing a server or a server class) simply instantiate a connection object and perform some additional operations.

class TCPConnection:

```
self.state = TCPClosed(self.sock)
```

```
def __init__(self, host, port):
    self.host = host
    self.port = port
    self.sock = socket.socket(socket.AF_INET, \
                              socket.SOCK_STREAM)
    self.sock.bind((host, port))
    self.sock.listen(5)
```

```
self.state = TCPListen(self.sock)
```

```
def OpenConnection(self):
    if state is TCPClosed:
        self.state = TCPListen
    if state is TCPListen:
        self.clisock = self.state.OpenConnection()
```

```
self.state = TCPEstablished(self.clisock)
```

```
def HandleConnection(self):
    if state is TCPClosed:
        raise CloseStateError
    if state is TCPListen:
        raise ListenStateError
    if state is TCPEstablished:
        self.state.HandleConnection(some_handler)
        self.state = TCPListen(self.sock)
```

```

def CloseConnection():
    self.state = TCPClosed(self.sock)
    self.state.CloseConnection()

```

As you can see in the example presented above, the behavior of the connection object depends on its state. The methods cannot be used in any order. When the object is instantiated, the `__init__()` performs the classic three steps initialization on server socket and brings the object to listen state. When in listen state, the client may call just `OpenConnection()` and `CloseConnection()` methods. After the connection is accepted (via `OpenConnection` - also see the code below, describing the states classes) the object is in the established state, when it deals with the other end of the connection. Now, the client may use `handleConnection()` to run the protocol. By default, the object returns itself to listen state and then `OpenConnection` must be called again to realize a new connection. When `CloseConnection()` is called, the server socket is closed and further operation on it will be discarded, thus the object is no longer of interest. This is a very basic approach, providing just a skeleton for a real `TCPConnection` object. `raise` statements are used in signaling the client over an improper use of the object's methods. These errors are constructed through inheritance from the `Exception` class. Below are described the four state classes:

```

class TCPState:

    def __init__(self, sock):
        self.sock = sock

    def OpenConnection(self): pass

    def HandleConnection(self): pass

    def CloseConnection(self): pass


class TCPClosed(TCPState):

    def CloseConnection(self, sock):
        sock.close()


class TCPListen(TCPState):

    def OpenConnection(self, sock):

        clisock, addr = sock.accept()
        return clisock


class TCPEstablished(TCPState):

    def HandleConnection(self, handler):
        thread.start_new_thread(handler, \
            (self.sock,))

```

All the concrete classes are inherited from the `TCPState` abstract class. This class provides a uniform interface for the other three. Please see that only the necessary methods were overridden. When another state is wanted, the state class is inherited from `TCPState`, methods are redefined and proper code is added to the `TCPConnection` class to include this state as well.

Advanced aspects concerning clients

In majority of this section 4 we focused on server design. This is true, because a server design is far much complicated related to client design. A server is running as demon in most of the cases, dealing with many clients, so it must be fast, reliable and must not be prone to errors. A client is maintaining a single connection with the server (this for the huge majority of clients), therefore the pure networking code is simpler and is not involving threading, selecting or forking. However, is recommended to apply object-oriented programming in this case or even a design pattern (the presented state design pattern is easy applicable to a client), but no fork, thread or select. Life is not so hard when one have to design the networking part of a client.

Chapter 5: HTTP protocol

HTTP is, beside electronic mail, the most popular protocol on Internet today. Please refer [8] and [9] for detailed informations about this protocol. HTTP is implemented in web servers like Apache or in browsers like Netscape or Mosaic. Python provides a few modules to access this protocol. We are mentioning a few of them: webbrowser, urllib, cgi, urllib, urllib2, urlparse and three modules for already build HTTP servers: BaseHTTPServer, SimpleHTTPServer and CGIHTTPServer. Is provided the Cookie module for cookies support.

The webbrowser module offers a way in displaying and controlling HTML documents via a web browser (this web browser is system dependent). Using the cgi module the user may be able to write complex CGI scripts, replacing "standard" CGI scripting languages as Perl. urllib is a module implementing the client side of the HTTP protocol. The urllib is module that provides a high-level interface for retrieving web-based content across World Wide Web. This module (and urllib2 too) uses the previous mentioned module, urllib. As it's name says, the urlparse module is used for parsing URLs into components.

CGI module

CGI (Common Gateway Interface) is a mean through one may send information from a HTML page (i.e. using the <form>...</form> structures) to a program or script via a web server. The program receive this information in a suitable form { depending on web server { and perform a task: add user to a database, check a password, generate a HTML page in response or any other action. Traditionally, this scripts live in the server's special directory 'cgi-bin', whom path must be set in the server configuration file.

Build a simple CGI script

When the user is calling a CGI script, a special string is passed to the server specifying the script, parameters (fed from a form or an URL with query string) and other things. The server is passing all this stu to the mentioned script which is supposed to understand what's happening, process the query and execute an action. If the script is replying something to client (all the print statements, the script output respectively), it will send the data chunk through the web server. Now is clear why this interface is called Common Gateway { every transaction between a client and a CGI script is made using the web server, more speci c, using the interface between the server and the script.

A simple script which simply displays a message on the screen is presented bellow:

```
print 'Content-Type: text/html' print
```

```
print 'Hello World!'
```

First two lines are mandatory: these lines tells the browser that the following content must be interpreted as HTML language. The print statement is used to create a blank line. Now, when the header is prepared, we may send to the client any output. If the output will contain HTML tags, they will be interpreted as a normal web page. Save this script in the 'cgi-bin' directory and set the permissions for it as world readable and executable. Make the appropriate modifications to the server's configuration file. Calling 'http://your.server.com/cgi-bin/your-script.py' you should see a small 'Hello World!' placed in the top left corner of your browser.

Let's modify the script to display the same message centered and with a reasonable size. We are going to use the <title>,<h1> and <center> HTML tags:

```
print 'Content-Type: text/html' print
```

```
print '<title>CGI Test Page</title>'
print '<center><h1>Hello World!</h1></center>'
```

Now the road is open. Try to generate your own HTML pages with CGI scripts. You can insert pictures, Java Script or VBScript in you pages. If the browser supports them, no problem.

Using CGI module

This module is useful when one might want to read information submitted by a user through a form. The module name is (well!) cgi and must be used with an import cgi statement (see [4]). The module defines a few classes (some for backward compatibility) that can handle the web server's output in order to make it accessible to programmer. The recommended class is FieldStorage class though in some cases MiniFieldStorage class is used. The class instance is a dictionary-like object and we can apply the dictionary methods. Bellow is the HTML page and the Python code to extract the informations from the form elds:

```
<html>
<form name='myform' method='POST' action='/cgi-bin/mycgi.py'> <input type='text'
name='yourname' size='30'>
<input type='textarea' name='comment' width='30' height='20'> <input type='submit'>
</form>
</html>

import cgi

FormOK = 0;
MyForm = cgi.FieldStorage();
if MyForm.has_key('yourname') and MyForm.has_key('comment'): FormOK = 1

print 'Content-Type: text/html' print

if FormOK:
    print '<p>Your name: ', Myform['yourname'].value, '<br>' print '<p>You comment: ',
    MyForm['comment'].value
else:
    print 'Error: fields have not been filled.'
```

As you can see in the example, MyForm is an instance of the FieldStorage class and will contain the name-value pairs described in the <form> structure in the HTML page: the text field with name=yourname and textarea with name=comment. Applying has key method to MyForm object one may be able to check if a particular eld have been lled and then, with MyForm['`name"].value can check the value of that field. Once the values have been assigned to some internal variables, the programmer may use them as any trivial variable. We present now a small example that read a form submitted by a user (containing the user name, address and phone number) and write all the values in a file.

```
<html>
<form name='personal_data' action='/cgi-bin/my-cgi.py' method='POST'> <p>Insert your name
.&nbsp;<input type='text' name='name' size='30'> <p>Insert your address:&nbsp;<input type='text'
name='address' size='30'> <p>Phone:&nbsp;<input type='text' name='tel' size='15'>
<p><input type='submit'> </form>
```

```
</html>
```

```
import cgi
```

```
MyForm = cgi.FieldStorage()
```

```
we suppose that all the fields have been filled  
and there is no need to check them
```

```
extracting the values from the form
```

```
Name = MyForm['name'].value
```

```
Address = MyForm['address'].value
```

```
Phone = myForm['tel'].value
```

```
# opening the file for writing
```

```
File = open('/my/path/to/file', 'r+')
```

```
File.write('Name: ' + Name)
```

```
File.write('Address: ' + Address)
```

```
File.write('Phone: ' + Phone)
```

```
File.close
```

```
print 'Content-Type: text/html' print
```

```
print 'Form submitted...ok.'
```

Configuring Apache on Linux for using with CGI scripts

This section is a brief troubleshooter for those which have some difficulties on setting up a script. If you are using Apache on Linux or whatever *NIX system you should see an error with the code 500 { Internal Server Error. Another error is File Not Found but I think that it's too trivial to discuss this here. If you receive a 500 error code here is a list of 'to do' in order to make your script working:

Check if the interpreter is specified on the first line of your script:

```
#!/usr/bin/env python
```

Please check if the interpreter is world readable and executable.

Check if the files accessed by the script are readable and executable by 'others' because, due to security reasons, Apache is running your script as user 'nobody'.

Make sure that in httpd.conf exist the following lines:

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
```

and

```
<Directory "/var/www/cgi-bin">    AllowOverride  
    None  
    Options ExecCGI Order allow,deny  
    Allow from all
```


</Directory>

The path to your 'cgi-bin' directory may vary, depending on how you have set the DocumentRoot option in the same file.

Also it's a good practice to run the script into a console, the interpreter printing a traceback in case any error occurs, this being the simplest way to correct syntax errors. For any other solutions and tips please refer [4].

Chapter 6: Common protocols

Designing Telnet applications

Telnet is a simple protocol that emulate a terminal over a networked link. With telnet you can access a foreign machine on a specified port and then send and receive data over the established link without any restrictions. It is possible to access via telnet clients a web server, a POP3 server, a mail server and so on. The purpose of the telnetlib is to offer a simple way to design a telnet client. This client may be used in further applications as the support for communication between two machines, using some protocol. To understand better the previous assumptions, please consider the next two examples:

```
$ telnet localhost
```

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
```

login:

This is a trivial telnet session. The remote machine will request you a user name and password (in case that the telnet service is up and running on that host and you are allowed to access it) and then will drop you in a shell. The main disadvantage: the communication is not confidential: telnet is transmitting everything in clear text. Let's consider the second example:

```
$ telnet localhost 80
```

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
```

```
GET /
...
```

In this case we try to access a web server (port 80) on some location. To receive a web page GET method is used followed by the name of the page (in this case / is the root page, commonly named index.html). We did not specify the complete GET request, mentioning the protocol and other things (if someone is interested on this topic check [9]).

Conclusion: the same client but different protocols. When we used the GET method in the second example, the telnet client was our network support and the protocol was emulated typing it. The core idea is to use a telnet client as support for the networking part while the user is writing (or using) its own protocol over this link.

Python is implementing the Telnet protocol with the telnetlib module which offers a class called Telnet(host, port). The instance of this class is a telnet object with methods for opening and closing a connection, reading and writing data, checking the status of the link and debugging the script. The previous examples are translated now in Python code using the telnetlib:

```
import telnetlib
```

```
telnet = telnetlib.Telnet('localhost')
telnet.read_until('login: ')
user = raw_input('Login: ')
telnet.write(user + '\n')
telnet.read_until('Password: ')
passwd = raw_input('Password: ')
telnet.write(passwd + '\n')
```

now the user may send commands
and read the proper outputs

A telnet object is created (in this case telnet) and the output of the server is read until the string 'login: ' is reached. Now the server is waiting for the login name, which is supplied reading it from the keyboard and then passing it through a variable to the write() method. The same algorithm is used for submitting the password. Once the user is authenticated it is possible to send commands and read the output sent by the server. The second example is a simple retrieval of a web page from a certain address, save it to a file and display it using the webbrowser module.

```
import telnetlib import webbrowser

telnet = telnetlib.Telnet('localhost', 80) telnet.read_until('^.\n')

telnet.write('GET /') html_page = telnet.read_all()

html_file = open('/some/where/file', 'w')
html_file.write(html_page) html_file.close()

webbrowser.open('/some/where/file')
```

The code above will read the index page from the specified location, will save it to a file and then, through the webbrowser module will display it into a browser. For a complete list of commands described in the telnetlib module please refer to [4].

File Transfer Protocol

Python currently implements this protocol using the ftplib module. This module is very useful when one may want to write a script for automatic retrieval of files or mirroring ftp sites on a daily (hourly, weekly) basis. The module is a simple translation in Python code, as methods of the FTP class, of the usual commands during a ftp session: get, put, cd, ls etc.

As was stated before, this module implements the FTP class. To test a ftp session, open a terminal, enter the Python interactive shell and type commands as below:

```
import ftplib
ftp_session = ftplib.FTP('ftp.some.host.com')
ftp_session.login()
'230 Guest login ok, access restriction apply.'
>>> ftp_session.dir()

....

>>> ftp_session.close()
```

The ftp_session is the instance of the FTP class. There are currently two methods implemented to open a connection to a given server: specifying the host name and port when instantiating the ftp object or applying the open(host, port) method for an unconnected ftp object.

```
import ftplib
ftp_session = ftplib.FTP()
ftp_session.open('ftp.some.host.com')

....
```

After the connection is realized the user must submit its login name and password (in this case was used the 'anonymous' user name and user@host password). The server will send back a string stating if it

accepts or reject the user's login and password. After the user has successfully entered the site, the following methods may be useful:

`pwd()` for listing the pathname of the current directory, `cwd(pathname)` for changing the directory and `dir(argument[...])` to list the content of the current (or argument) directory, methods for navigation and browsing through the directories.

`rename(fromname, toname)`, `delete(filename)`, `mkd(pathname)`, `rmd(pathname)` and `size(filename)` methods (all of them are self-explanatory) to manage files and directories inside an ftp site.

`retrbinary(command, callback[, maxblocksize[, rest]])` method used to retrieve a binary file (or to retrieve a text file in binary mode). `command` is a 'RETR lename' (see [4]), `callback` is a function called for each data block received. To receive a file or the output of a command in text mode `retrlines(command[, callback])` method is provided.

Many others commands are provided and you may find a complete description in Python Library Reference [4].

SMTP protocol

The Simple Mail Transfer Protocol described in [11], is a protocol that may be used to send email to any machine on the Internet with a SMTP daemon listening on port 25 (default port for SMTP service). Python implements the client side of this protocol (one may only send but not receive mail) in the `smtplib` module. As for `telnetlib` and `ftplib` modules, is defined a `SMTP([host[, port]])` class which instance, an SMTP object, has several methods that emulate SMTP. The module also offers a rich set of exception, useful in different situations. An exhaustive list of all the exceptions defined in this module is presented below for the sake of completeness:

SMTPException

`SMTPServerDisconnected`

`SMTPResponseException`

`SMTPSenderRefused`

`SMTPRecipientsRefused`

`SMTPDataError`

`SMTPConnectionError`

`SMTPHeloError`

Let's see in a short example, how do we use the methods available in this module to send a simple test mail to a remote machine named 'foreign.host.com' for user 'xxx':

```
import smtplib, sys

s = smtplib.SMTP()
s.set_debuglevel(1)
s.connect('foreign.host.com') #default port 25
s.helo('foreign.host.com')

ret_code = s.verify('xxx@foreign.host.com') if ret_code[0] == 550:
    print 'Error, no such user.' sys.exit(1)
```

```
s.sendmail('user@localhost', 'xxx@foreign.host.com', 'Test message!') s.quit()
```

The script begins by instantiating the SMTP() class, here without any parameters. s.connect('foreign.host.com') is used to connect the client to the remote SMTP server. After the connection is realized our client will send an HELO message. Here it is useful to check if the server accepts this message (with a try... clause using the SMTPHeloError exception). There is no reason for the server to reject the HELO greeting but it is useful to make such an error checking. With s.verify('xxx@foreign.host.com') the script checks if the receiver address exists on that server. ret_code is a variable (list) which will contain the server reply after the instruction is executed, containing two elements: an integer, meaning the reply code and a string for human consumption. In case the user exists on that machine, the code is 250 and in case of error we receive a 550. The most important method is sendmail(fromaddr, toaddr, message[, mail options, recipient options]), which is the real sender of the message. Because we are such polite people, we are doing a quit() before effectively exiting the script. This will assure that everything is alright when leaving the SMTP session.

Chapter 7: Getting Started With Python in Windows

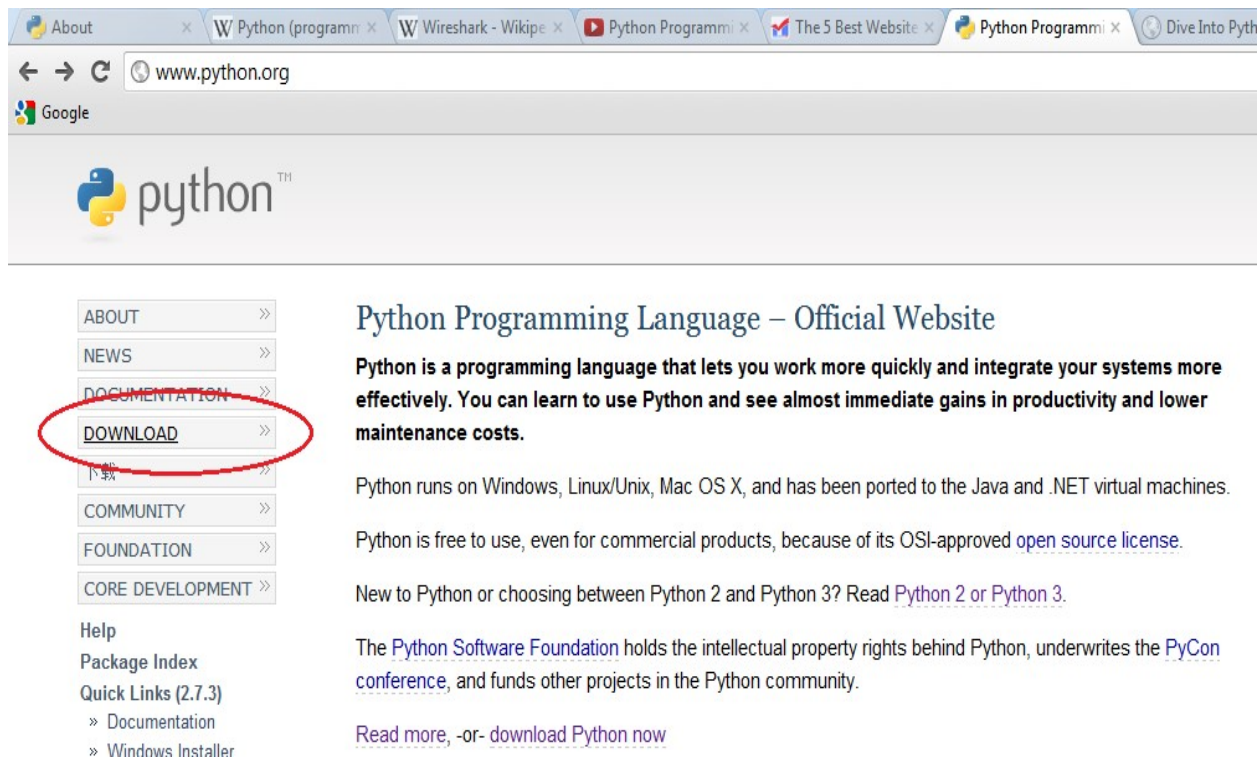
Python is a general purpose, high level programming language that is used in a variety of application domains. The Python language has a very clear and expressive syntax as well as a large and comprehensive library. Although Python is often used as a scripting language, it can also be used in a wide range of non-scripting contexts. It's available for all major Operating Systems: Windows, Linux/Unix, OS/2, Mac, Amiga, among others. Python is free to use, even for commercial products, because of its OSI-approved open source license.

Python 2 or Python 3?

Python has two standard versions, Python 2 and Python 3. The current production versions (July 2012) are Python 2.7.3 and Python 3.2.3. *Python 2.7 is the status quo*. We recommend you use Python 2.7 for completing the assignments.

Installing Python

Python can be downloaded directly from the official website <http://www.python.org/>. This is the program that is used to write all your python code. On the left side of the website there is a download section



Clicking the download link will present you with two versions of Python, namely *python 2* and *python 3*; you can also choose the version specific to your operating system. Most popular Linux distributions come

with Python in the default installation. Mac OS X 10.2 and later includes a command-line version of Python, although you'll probably want to install a version that includes a more Mac-like graphical interface.

Installing Python on Windows

Double-click the installer, Python-2.xxx.yyy.exe. The name will depend on the version of Python available when you read this.

Select run.

Step through the installer program.

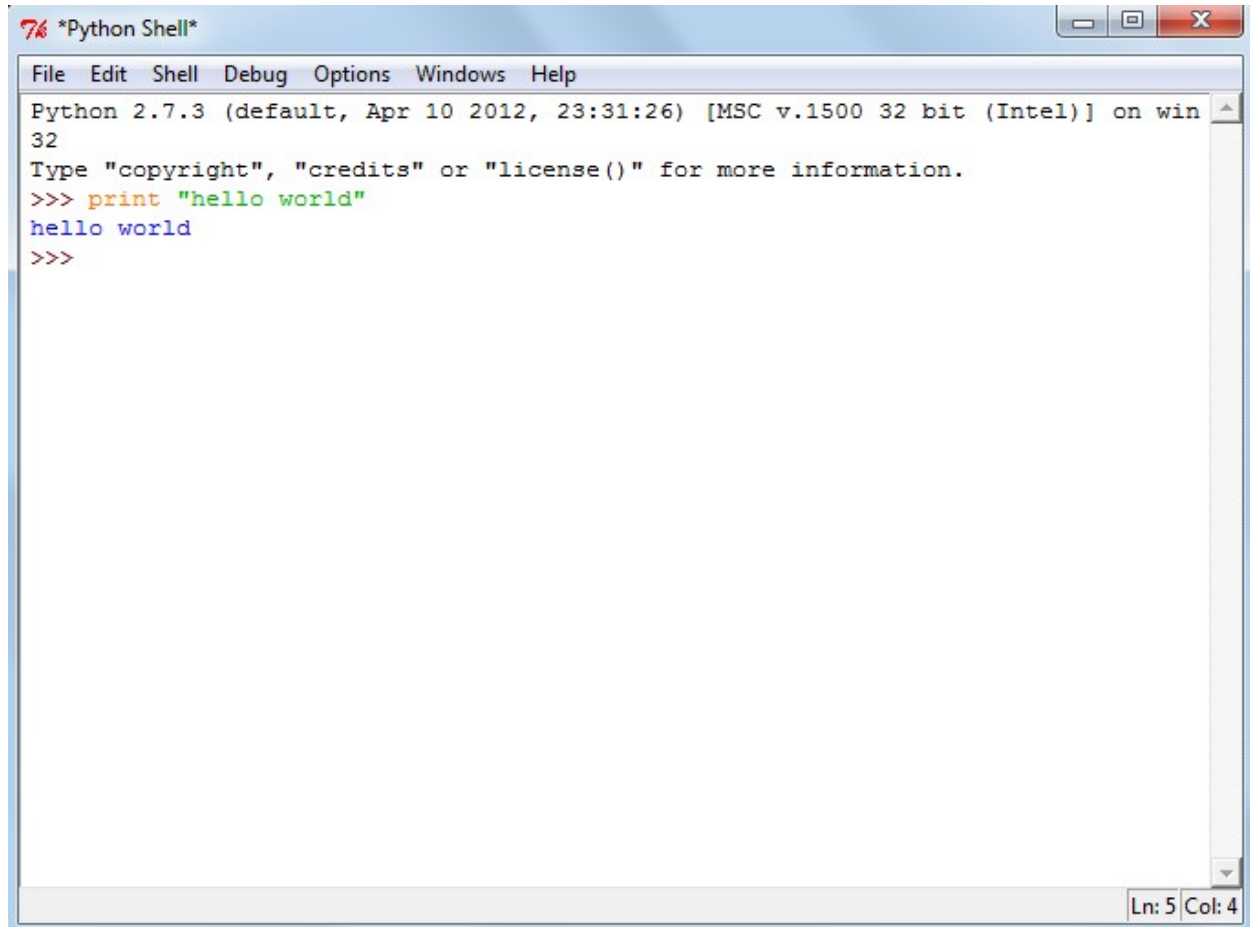
If disk space is tight, you can deselect the HTMLHelp file, the utility scripts (Tools/), and/or the test suite (Lib/test/).

If you do not have administrative rights on your machine, you can select Advanced Options, then choose Non-Admin Install. This just affects where Registry entries and Start menu shortcuts are created.

If you see the following that means the installation is complete.



After the installation is complete, close the installer and select Start->Programs->Python 2.3>IDLE (Python GUI). You'll see something like the following:

A screenshot of the Python Shell window. The title bar reads '*Python Shell*'. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main text area shows the following text: 'Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32', 'Type "copyright", "credits" or "license()" for more information.', '>>> print "hello world"', 'hello world', and '>>>'. The status bar at the bottom right indicates 'Ln: 5 Col: 4'.

Other Window Installation Options

ActiveState makes a Windows installer for Python called ActivePython, which includes a complete version of Python, an IDE with a Python-aware code editor, plus some Windows extensions for Python that allow complete access to Windows-specific services, APIs, and the Windows Registry. ActivePython is freely downloadable, although it is not open source. You recommend you use this for writing more complicated programs.

Download ActivePython from <http://www.activestate.com/Products/ActivePython/>. If you are using Windows 95, Windows 98, or Windows ME, you will also need to download and install [Windows Installer 2.0](#) before installing ActivePython.

Installing Python On Mac

The latest version of Mac OS X, Lion, comes with a command line version preinstalled. This version is great for learning but is not good for development. The preinstalled version may be slightly out of date, it does not come with an XML parser, also Apple has made significant changes that can cause hidden bugs.

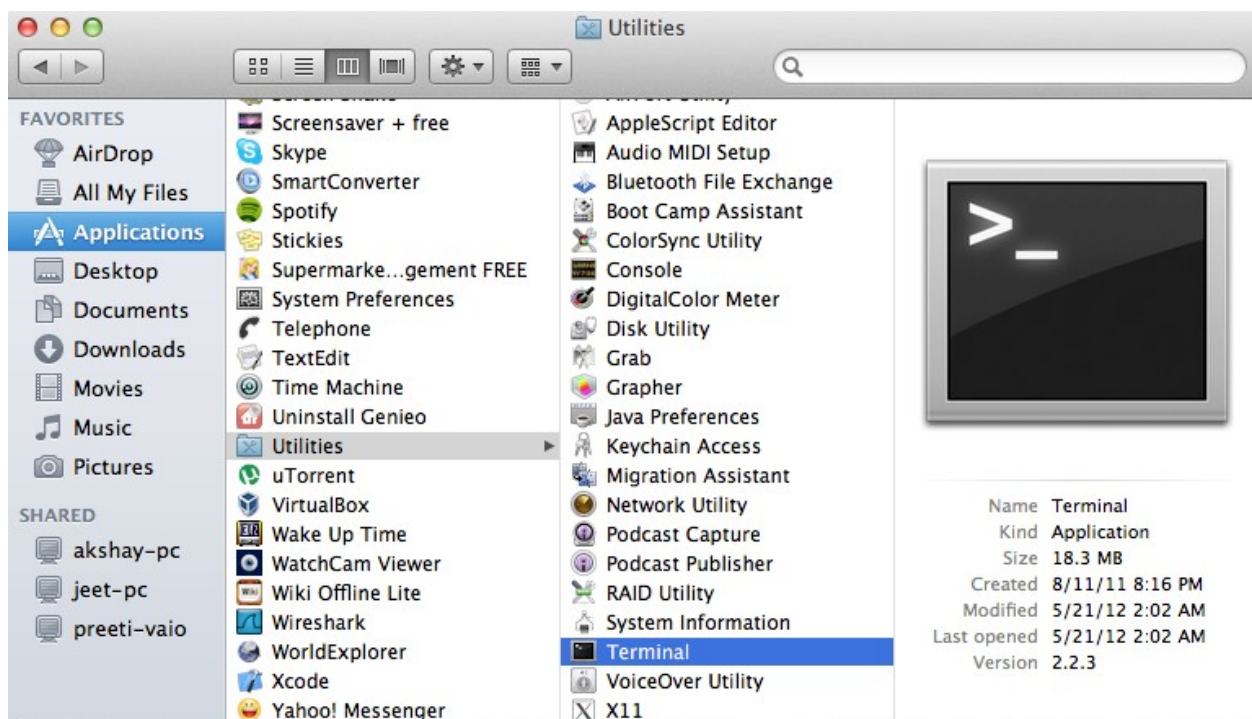
Rather than using the preinstalled version, you'll probably want to install the latest version, which also comes with a graphical interactive shell.

Running the Preinstalled Mac Version

Follow these steps in order to use the preinstalled version.

Go to Finder->Applications->Utilities.

Double click Terminal to get a command line.



Type **python** at the command prompt

Now you can try out some basic codes here

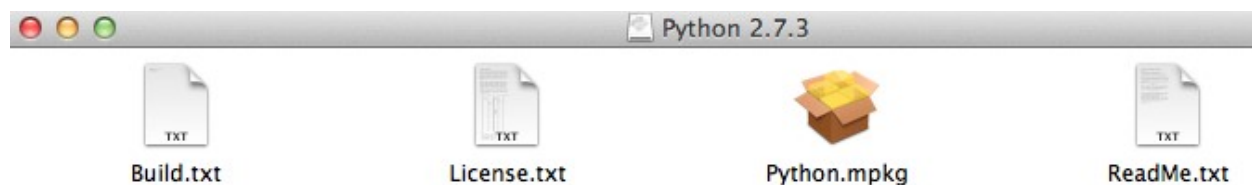
```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012, 20:32:06)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> print "hello world"
hello world
>>> _
```

Installing the Latest Version on the Mac

As said earlier Python comes preinstalled on Mac OS X , but due to Apple's release cycle, its often a year or two old. The "MacPython" community highly recommends you to upgrade your Python by downloading and installing a newer version.

Go to <http://www.python.org/download/> and download the version suitable for your system from among a list of options.

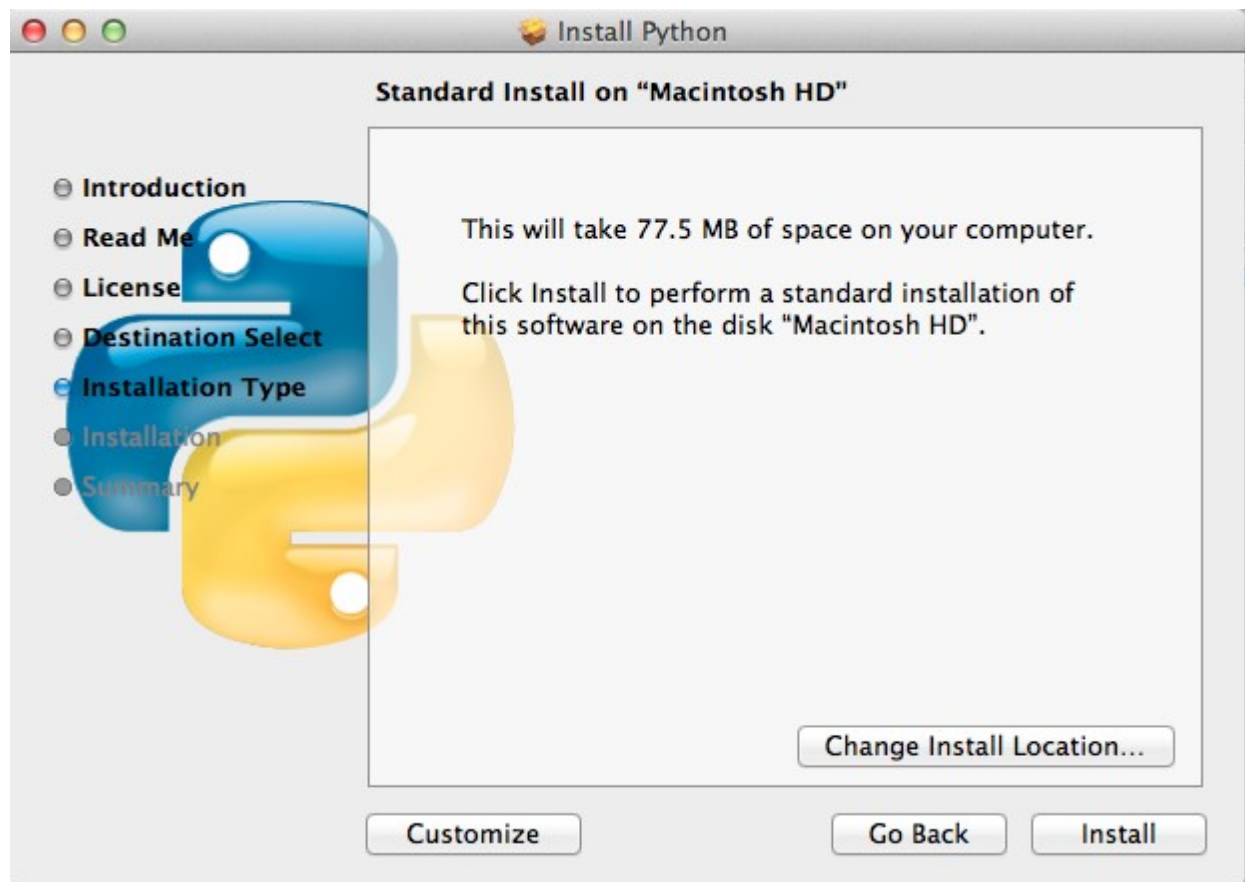
The downloaded file should look like this

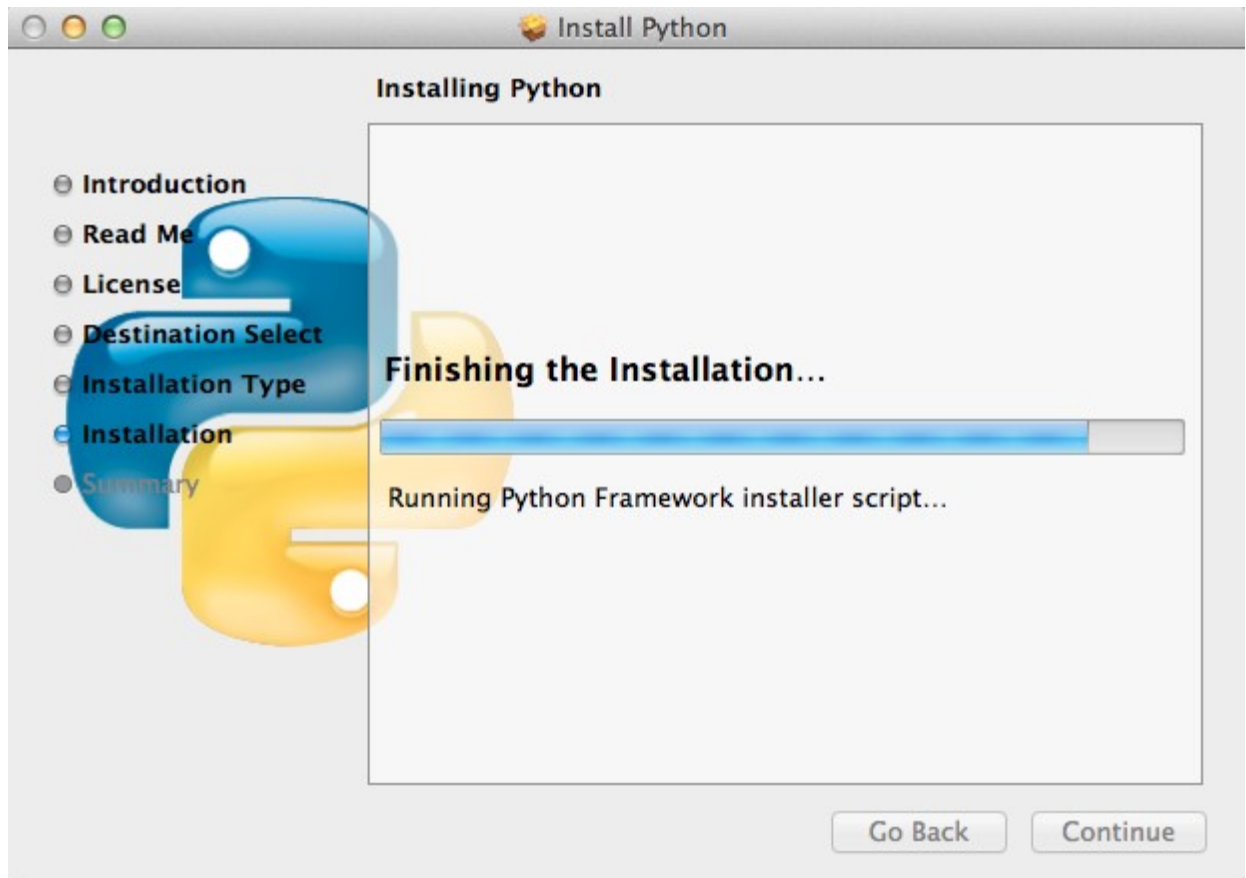


Double click the "Python.mpkg" file. The installer may prompt you for your administrative username and password.

Step through the installer program.

You can choose the location at which it is to be installed.





After the installation is complete, close the installer and open the Applications folder , search for Python and you'll see the Python IDLE i.e. the standard GUI that comes with the package.

Alternative Packages for Mac OS X

[ActiveState ActivePython](#) (commercial and community versions, including scientific computing modules). ActivePython also includes a variety of modules that build on the solid core.

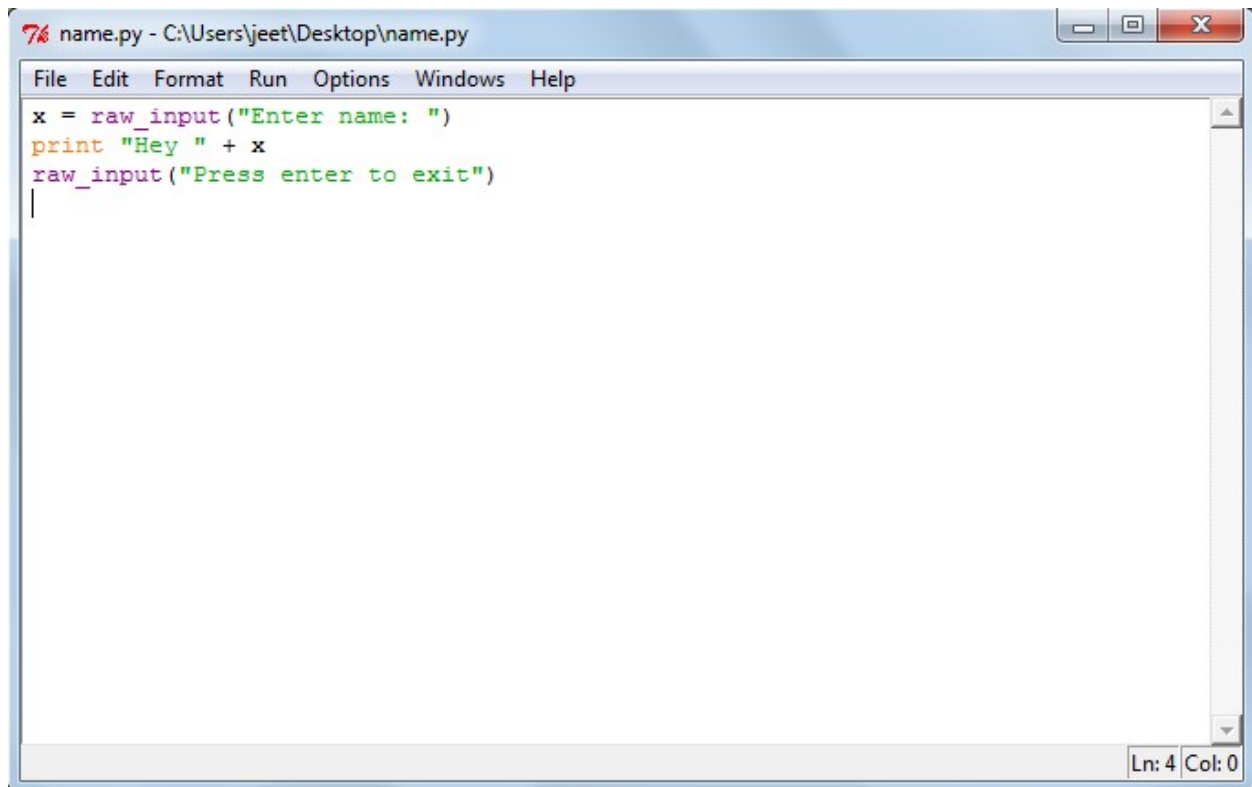
[Enthought Python Distribution](#) The Enthought Python Distribution provides scientists with a comprehensive set of tools to perform rigorous data analysis and visualization.

Example of a Basic Python Program

The interface “Idle” that we opened so far is only useful for testing out basic python commands or can otherwise be used as a calculator, it basically means the program cannot be saved this way.

To save a program and execute it we need to follow the following instruction: On the top left corner of “Idle” select File -> New Window.

The new window that pops out will allow you to save and execute your python programs. You can write your python code in this window. Try the following:

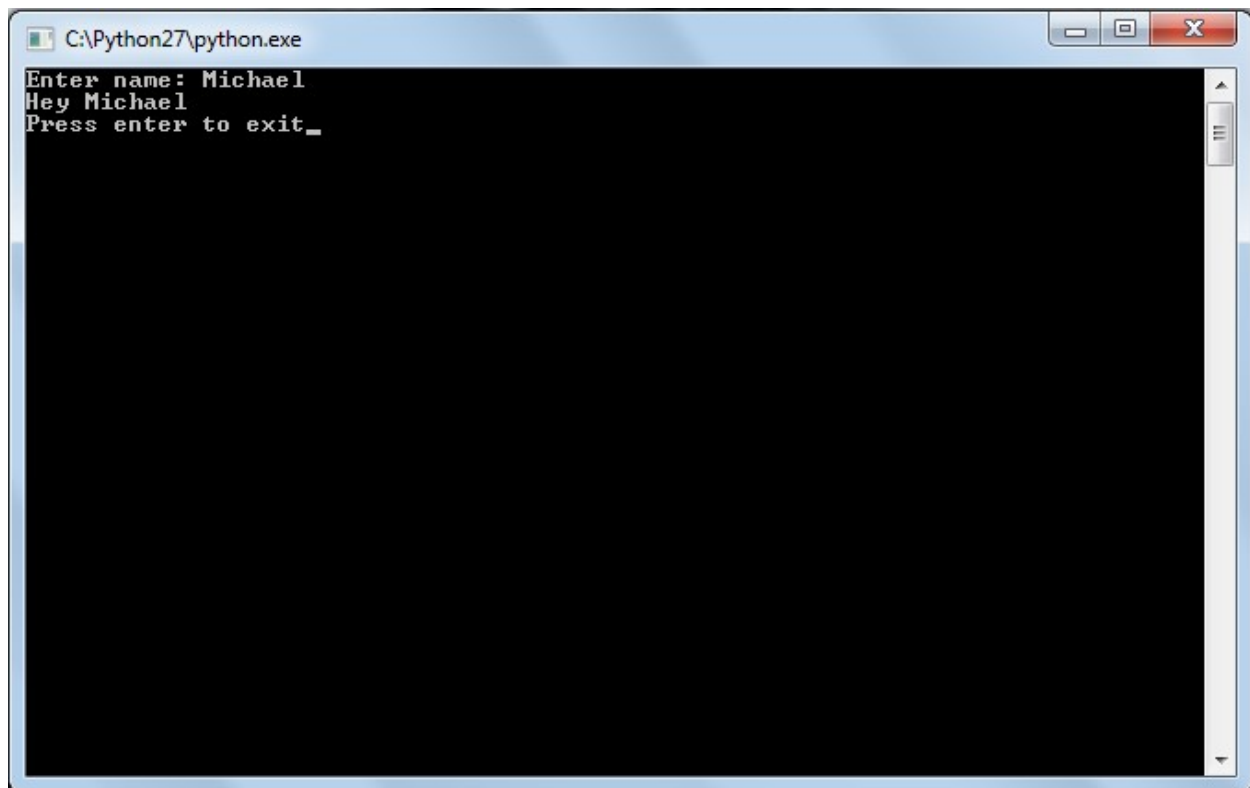


```
7% name.py - C:\Users\jeet\Desktop\name.py
File Edit Format Run Options Windows Help
x = raw_input("Enter name: ")
print "Hey " + x
raw_input("Press enter to exit")
|
Ln: 4 Col: 0
```

We cannot run this program without saving it. A saved python file has an icon that looks like this



You run the program by pressing F5 or Run-> Run Module. You can also run the program by simply double clicking the file icon. When you open a saved file to run the program, you should see:



```
C:\Python27\python.exe
Enter name: Michael
Hey Michael
Press enter to exit_
```

Learning Python Programming

Python is easy to learn, easy to use and very powerful. There are a lot of web resources for learning the language, most of which are entirely free. We recommend *Sthurlow.com's* A Beginner's Python Tutorial:

<http://www.sthurlow.com/python/>

Socket Programming Assignment 1: Web Server

In this lab, you will learn the basics of socket programming for TCP connections in Python: how to create a socket, bind it to a specific address and port, as well as send and receive a HTTP packet. You will also learn some basics of HTTP header format.

You will develop a web server that handles one HTTP request at a time. Your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an HTTP response message consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server, the server should send an HTTP "404 Not Found" message back to the client.

Code

Below you will find the skeleton code for the Web server. You are to complete the skeleton code. The places where you need to fill in code are marked with **#Fill in start** and **#Fill in end**. Each place may require one or more lines of code.

Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. Run the server program. Determine the IP address of the host that is running the server (e.g., 128.238.251.26). From another host, open a browser and provide the corresponding URL. For example:

http://128.238.251.26:6789/HelloWorld.html

'HelloWorld.html' is the name of the file you placed in the server directory. Note also the use of the port number after the colon. You need to replace this port number with whatever port you have used in the server code. In the above example, we have used the port number 6789. The browser should then display the contents of HelloWorld.html. If you omit ":6789", the browser will assume port 80 and you will get the web page from the server only if your server is listening at port 80.

Then try to get a file that is not present at the server. You should get a "404 Not Found" message.

What to Hand in

You will hand in the complete server code along with the screen shots of your client browser, verifying that you actually receive the contents of the HTML file from the server.

Skeleton Python Code for the Web Server

```
#import socket module from socket import *

serverSocket = socket(AF_INET, SOCK_STREAM)

#Prepare a sever socket

#Fill in start

#Fill in end while True:

    #Establish the connection    print 'Ready to serve...'    connectionSocket, addr = #Fill in start

#Fill in end        try:

            message = #Fill in start        #Fill in end            filename =

message.split()[1]                f = open(filename[1:])

outputdata = #Fill in start    #Fill in end

    #Send one HTTP header line into socket

    #Fill in start    #Fill in end

    #Send the content of the requested file to the client        for i in range(0,

len(outputdata)):

        connectionSocket.send(outputdata[i])

connectionSocket.close()    except IOError:

    #Send response message for file not found

    #Fill in start        #Fill in end
```

```
#Close client socket
```

```
#Fill in start    #Fill in end
```

```
serverSocket.close()
```

Optional Exercises

Currently, the web server handles only one HTTP request at a time. Implement a multithreaded server that is capable of serving multiple requests simultaneously. Using threading, first create a main thread in which your modified server listens for clients at a fixed port. When it receives a TCP connection request from a client, it will set up the TCP connection through another port and services the client request in a separate thread. There will be a separate TCP connection in a separate thread for each request/response pair.

Instead of using a browser, write your own HTTP client to test your server. Your client will connect to the server using a TCP connection, send an HTTP request to the server, and display the server response as an output. You can assume that the HTTP request sent is a GET method.

The client should take command line arguments specifying the server IP address or host name, the port at which the server is listening, and the path at which the requested object is stored at the server. The following is an input command format to run the client.

```
client.py server_host server_port filename
```


Socket Programming Assignment 2: UDP

In this lab, you will learn the basics of socket programming for UDP in Python. You will learn how to send and receive datagram packets using UDP sockets and also, how to set a proper socket timeout. Throughout the lab, you will gain familiarity with a Ping application and its usefulness in computing statistics such as packet loss rate.

You will first study a simple Internet ping server written in the Python, and implement a corresponding client. The functionality provided by these programs is similar to the functionality provided by standard ping programs available in modern operating systems. However, these programs use a simpler protocol, UDP, rather than the standard Internet Control Message Protocol (ICMP) to communicate with each other. The ping protocol allows a client machine to send a packet of data to a remote machine, and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines.

You are given the complete code for the Ping server below. Your task is to write the Ping client.

Server Code

The following code fully implements a ping server. You need to compile and run this code before running your client program. *You do not need to modify this code.*

In this server code, 30% of the client's packets are simulated to be lost. You should study this code carefully, as it will help you write your ping client.

```
# UDPPingerServer.py
# We will need the following module to generate randomized lost packets import
random from socket import *

# Create a UDP socket
# Notice the use of SOCK_DGRAM for UDP packets
serverSocket = socket(AF_INET, SOCK_DGRAM) #
Assign IP address and port number to socket
serverSocket.bind(('', 12000))

while True:
    # Generate random number in the range of 0 to 10 rand
    = random.randint(0, 10)
    # Receive the client packet along with the address it is coming from message,
    address = serverSocket.recvfrom(1024)
    # Capitalize the message from the client message
    = message.upper()
    # If rand is less is than 4, we consider the packet lost and do not respond if rand < 4:
    continue
```

```
# Otherwise, the server responds
serverSocket.sendto(message, address)
```

The server sits in an infinite loop listening for incoming UDP packets. When a packet comes in and if a randomized integer is greater than or equal to 4, the server simply capitalizes the encapsulated data and sends it back to the client.

Packet Loss

UDP provides applications with an unreliable transport service. Messages may get lost in the network due to router queue overflows, faulty hardware or some other reasons. Because packet loss is rare or even non-existent in typical campus networks, the server in this lab injects artificial loss to simulate the effects of network packet loss. The server creates a variable randomized integer which determines whether a particular incoming packet is lost or not.

Client Code

You need to implement the following client program.

The client should send 10 pings to the server. Because UDP is an unreliable protocol, a packet sent from the client to the server may be lost in the network, or vice versa. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should get the client wait up to one second for a reply; if no reply is received within one second, your client program should assume that the packet was lost during transmission across the network. You will need to look up the Python documentation to find out how to set the timeout value on a datagram socket.

Specifically, your client program should

(1) send the ping message using UDP (Note: Unlike TCP, you do not need to establish a connection first, since UDP is a connectionless protocol.) (2) print the response message from server, if any

calculate and print the round trip time (RTT), in seconds, of each packet, if server responses otherwise, print "Request timed out"

During development, you should run the UDPPingerServer.py on your machine, and test your client by sending packets to *localhost* (or, 127.0.0.1). After you have fully debugged your code, you should see how your application communicates across the network with the ping server and ping client running on different machines.

Message Format

The ping messages in this lab are formatted in a simple way. The client message is one line, consisting of ASCII characters in the following format:

Ping *sequence_number* *time*

where *sequence_number* starts at 1 and progresses to 10 for each successive ping message sent by the client, and *time* is the time when the client sends the message.

What to Hand in

You will hand in the complete client code and screenshots at the client verifying that your ping program works as required.

Optional Exercises

Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).

Another similar application to the UDP Ping would be the UDP Heartbeat. The Heartbeat can be used to check if an application is up and running and to report one-way packet loss. The client sends a sequence number and current timestamp in the UDP packet to the server, which is listening for the Heartbeat (i.e., the UDP packets) of the client. Upon receiving the packets, the server calculates the time difference and reports any lost packets. If the Heartbeat packets are missing for some specified period of time, we can assume that the client application has stopped. Implement the UDP Heartbeat (both client and server). You will need to modify the given UDPPingerServer.py, and your UDP ping client.

Socket Programming Assignment 3: SMTP

By the end of this lab, you will have acquired a better understanding of SMTP protocol. You will also gain experience in implementing a standard protocol using Python.

Your task is to develop a simple mail client that sends email to any recipient. Your client will need to connect to a mail server, dialogue with the mail server using the SMTP protocol, and send an email message to the mail server. Python provides a module, called `smtplib`, which has built in methods to send mail using SMTP protocol. However, we will not be using this module in this lab, because it hides the details of SMTP and socket programming.

In order to limit spam, some mail servers do not accept TCP connection from arbitrary sources. For the experiment described below, you may want to try connecting both to your university mail server and to a popular Webmail server, such as a AOL mail server. You may also try making your connection both from your home and from your university campus.

Code

Below you will find the skeleton code for the client. You are to complete the skeleton code. The places where you need to fill in code are marked with **#Fill in start** and **#Fill in end**. Each place may require one or more lines of code.

Additional Notes

In some cases, the receiving mail server might classify your e-mail as junk. Make sure you check the junk/spam folder when you look for the e-mail sent from your client.

What to Hand in

In your submission, you are to provide the complete code for your SMTP mail client as well as a screenshot showing that you indeed receive the e-mail message.

Skeleton Python Code for the Mail Client from socket import *

```
msg = "\r\n I love computer networks!"
```

```
endmsg = "\r\n.\r\n"
```

```
# Choose a mail server (e.g. Google mail server) and call it mailserver mailserver = #Fill in start
```

```
#Fill in end
```

```
# Create socket called clientSocket and establish a TCP connection with mailserver
```

```
#Fill in start
```

```
#Fill in end
```

```
recv =
```

```
clientSocket.recv(1024)
```

```
print recv if recv[:3] !=
```

```
'220':
```

```
    print '220 reply not received from server.'
# Send HELO command and print server
response. heloCommand = 'HELO Alice\r\n'
clientSocket.send(heloCommand)  recv1 =
clientSocket.recv(1024)
print recv1 if recv1[:3] != '250':    print '250 reply
not received from server.'
```

```
# Send MAIL FROM command and print server response.
```

```
# Fill in start
```

```
# Fill in end
```

```
# Send RCPT TO command and print server response.
```

```
# Fill in start
```

```
# Fill in end
```

```
# Send DATA command and print server response.
```

```
# Fill in start
```

```
# Fill in end
```

```
#      Send
message
data. # Fill in
start
```

```
# Fill in end
```

```
# Message ends with a single period.
```

```
# Fill in start
```

```
# Fill in end
```

```
# Send QUIT command and get server response.
```

```
# Fill in start
```

```
# Fill in end
```

Optional Exercises

Mail servers like Google mail (address: smtp.gmail.com, port: 587) requires your client to add a Transport Layer Security (TLS) or Secure Sockets Layer (SSL) for authentication and security reasons, before you send MAIL FROM command. Add TLS/SSL commands to your

existing ones and implement your client using Google mail server at above address and port.
Your current SMTP mail client only handles sending text messages in the email body. Modify your client such that it can send emails with both text and images.

Socket Programming Assignment 4: ICMP Pinger

In this lab, you will gain a better understanding of Internet Control Message Protocol (ICMP). You will learn to implement a Ping application using ICMP request and reply messages.

Ping is a computer network application used to test whether a particular host is reachable across an IP network. It is also used to self-test the network interface card of the computer or as a latency test. It works by sending ICMP "echo reply" packets to the target host and listening for ICMP "echo reply" replies. The "echo reply" is sometimes called a pong. Ping measures the round-trip time, records packet loss, and prints a statistical summary of the echo reply packets received (the minimum, maximum, and the mean of the round-trip times and in some versions the standard deviation of the mean).

Your task is to develop your own Ping application in Python. Your application will use ICMP but, in order to keep it simple, will not exactly follow the official specification in RFC 1739. Note that you will only need to write the client side of the program, as the functionality needed on the server side is built into almost all operating systems.

You should complete the Ping application so that it sends ping requests to a specified host separated by approximately one second. Each message contains a payload of data that includes a timestamp. After sending each packet, the application waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client assumes that either the ping packet or the pong packet was lost in the network (or that the server is down).

Code

Below you will find the skeleton code for the client. You are to complete the skeleton code. The places where you need to fill in code are marked with **#Fill in start** and **#Fill in end**. Each place may require one or more lines of code.

Additional Notes

In "receiveOnePing" method, you need to receive the structure ICMP_ECHO_REPLY and fetch the information you need, such as checksum, sequence number, time to live (TTL), etc.

Study the "sendOnePing" method before trying to complete the "receiveOnePing" method.

You do not need to be concerned about the checksum, as it is already given in the code.

This lab requires the use of raw sockets. In some operating systems, you may need administrator/root privileges to be able to run your Pinger program.

See the end of this programming exercise for more information on ICMP.

Testing the Pinger

First, test your client by sending packets to localhost, that is, 127.0.0.1.

Then, you should see how your Pinger application communicates across the network by pinging servers in different continents.

What to Hand in

You will hand in the complete client code and screenshots of your Pinger output for four

target hosts, each on a different continent.

Skeleton Python Code for the ICMP Pinger
from socket import *

```
import os
import
sys
import
struct
import
time
import
select
import
binascii
```

```
ICMP_ECHO_REQUEST = 8
```

```
def
checksum(
str):
    csum = 0
    countTo = (len(str) / 2) * 2
    count = 0    while count < countTo:        thisVal =
ord(str[count+1]) * 256 + ord(str[count])
    csum = csum + thisVal        csum =
csum & 0xffffffffL
    count = count + 2
    if countTo < len(str):        csum = csum +
ord(str[len(str) - 1])        csum = csum &
0xffffffffL
```

```
    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer    =    ~csum
answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer
def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout
```

```
    while
    e 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)    howLongInSelect = (time.time() -
```

```

startedSelect)
    if whatReady[0] == []: # Timeout        return
    "Request timed out."

    timeReceived = time.time()        recPacket, addr =
mySocket.recvfrom(1024)

    #Fill in start

    #Fetch the ICMP header from the IP packet

    #Fill in end

    timeLeft = timeLeft - howLongInSelect
    if timeLeft <= 0:        return "Request timed
out."
    def sendOnePing(mySocket, destAddr,
ID):
        # Header is type (8), code (8), checksum (16), id (16), sequence (16)

myChecksus
m = 0
    # Make a dummy header with a 0 checksum.
    # struct -- Interpret strings as packed binary data
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
    data = struct.pack("d", time.time())
    # Calculate the checksum on the data and the dummy header.    myChecksum =
checksum(header + data)

    # Get the right checksum, and put in the header    if
sys.platform == 'darwin':
        myChecksum = socket.htons(myChecksum) & 0xffff    #Convert 16-
bit integers from host to network byte order.
    else:
        myChecksum = socket.htons(myChecksum)

    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)    packet = header +
data
    mySocket.sendto(packet, (destAddr, 1)) # AF_INET address must be tuple, not str
#Both LISTS and TUPLES consist of a number of objects
#which can be referenced by their position number within the object
def doOnePing(destAddr, timeout):

```

```
icmp = socket.getprotobyname("icmp")
#SOCK_RAW is a powerful socket type.
For more details see: http://sock-raw.org/books/sock\_raw
```

#Fill in start

#Create Socket here

#Fill in end

```
myID = os.getpid() & 0xFFFF #Return the current process i
sendOnePing(mySocket, destAddr, myID) delay =
    receiveOnePing(mySocket, myID, timeout, destAddr)

mySocket.close()
return delay

def ping(host,
timeout=1):
    #timeout=1 means: If one second goes by without a reply from the server,
    #the client assumes that either the client's ping or the server's pong is lost
    dest = socket.gethostbyname(host)
    print
    "Pinging " + dest + " using Python:"
    print ""
    #Send ping requests to a server separated by approximately one second while 1 :
    delay = doOnePing(dest, timeout)
    print delay
    time.sleep(1)# one second
    return delay

ping("www.poly.edu")
```

Optional Exercises

Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).

Your program can only detect timeouts in receiving ICMP echo responses. Modify the Pinger program to parse the ICMP response error codes and display the corresponding error results to the user. Examples of ICMP response error codes are 0: Destination Network Unreachable, 1: Destination Host Unreachable.

Internet Control Message Protocol (ICMP)

ICMP Header

The ICMP header starts after bit 160 of the IP header (unless IP options are used).

Bits	160-167		168-175	176-183	184-191
160	Type		Code	Checksum	
192	I	D		Sequence	

Type - ICMP type.

Code - Subtype to the given ICMP type.

Checksum - Error checking data calculated from the ICMP header + data, with value 0 for this field.

ID - An ID value, should be returned in the case of echo reply.

Sequence - A sequence value, should be returned in the case of echo reply.

Echo Request

The echo request is an ICMP message whose data is expected to be received back in an echo reply ("pong"). The host must respond to all echo requests with an echo reply containing the exact data received in the request message.

Type must be set to 8.

Code must be set to 0.

The Identifier and Sequence Number can be used by the client to match the reply with the request that caused the reply. In practice, most Linux systems use a unique identifier for every ping process, and sequence number is an increasing number within that process. Windows uses a fixed identifier, which varies between Windows versions, and a sequence number that is only reset at boot time.

The data received by the echo request must be entirely included in the echo reply.

Echo Reply

The echo reply is an ICMP message generated in response to an echo request, and is mandatory for all hosts and routers.

Type and code must be set to 0.

The identifier and sequence number can be used by the client to determine which echo requests are associated with the echo replies.

The data received in the echo request must be entirely included in the echo reply.

References

- S. Leier et al: An Advanced 4.3BSD Interprocess Communication Tutorial
- A. Tanenbaum: Computer Networks 3rd-ed., Prentice-Hall, 1996
- E. Gamma et al: Design patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
- G. van Rossum: Python Library Reference
- G. van Rossum: Python Tutorial
- G. van Rossum: Python Language Reference
- G. McMillan: Socket Programming HOWTO
- ***: HTTP 1.0 - RFC1945
- ***: HTTP 1.1 - RFC2068
- ***: Telnet Protocol Specifications - RFC854
- ***: Simple Mail Transfer Protocol - RFC821
- ***: SMTP Service Extensions - RFC1869